Programmeerimise algkursus: listid

Overview of the lecture

- Linked list: a very common recursive data structure
- List vs array
- Stacks and queues
- Abstract data types
- Trees

Linked data structures

- A reference to one object can be stored in an instance variable of another object. The objects are then said to be "linked.“

- Complex data structured can be built by linking objects together. An especially interesting case occurs when an object contains a link to another object that belongs to the same class. In that case, the class is used in its own definition.

- Several important types of data structures are built using classes of this kind.

Recursive data structures

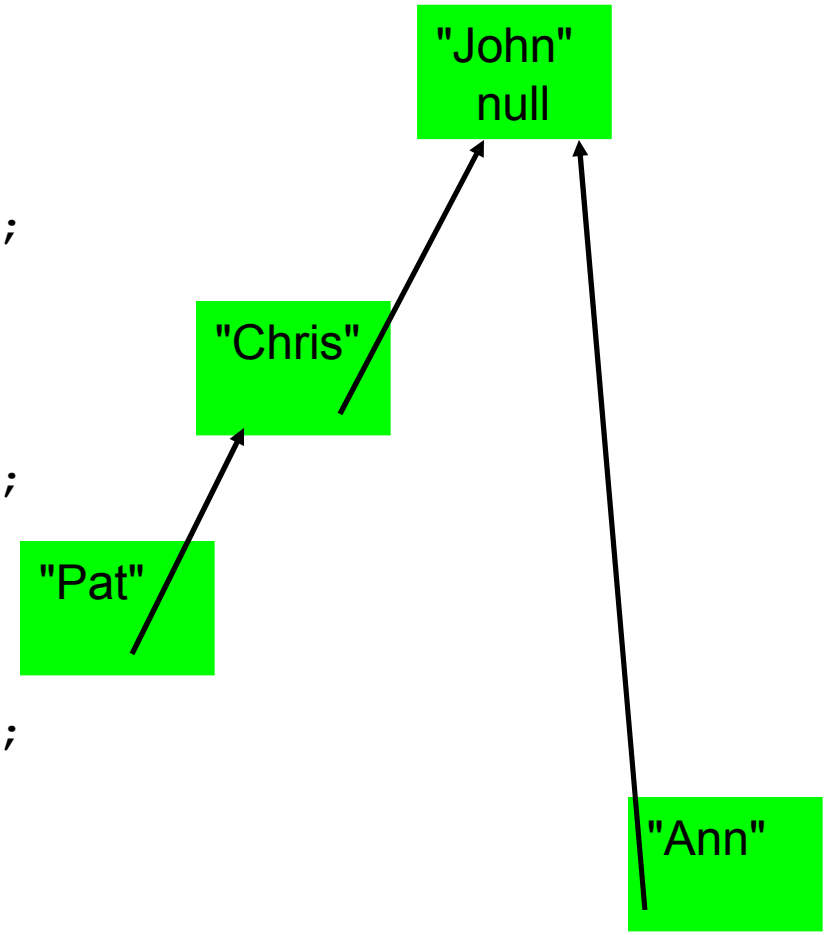
- Recursive data structures (class declarations) make it possible to create arbitrarily large data structures.

```
class Employee {
      // An object of type Employee holds data about
      //    one employee.
      String name;           // Name of the employee.
      Employee supervisor;   // The employee's supervisor.
        .
        .  // (Other instance variables and methods.)
        .
} // end class Employee
```

- If **emp** is a variable of type Employee, then **emp.supervisor** is another variable of type Employee, as well as **emp.supervisor.supervisor**
- If emp refers to the boss, then the value of emp.supervisor should be null to indicate the fact that the boss has no supervisor.

Example

```
Employee e1 = new Employee();
e1.name = "John";
e1.supervisor=null;
Employee e2 = new Employee();
e2.name = "Chris";
e2.supervisor = e1;
Employee e3 = new Employee();
e3.name = "Pat";
e3.supervisor = e2;
Employee e4 = new Employee();
e4.name = "Ann";
e4.supervisor = e2;
```



Count steps to the top-level boss: code example

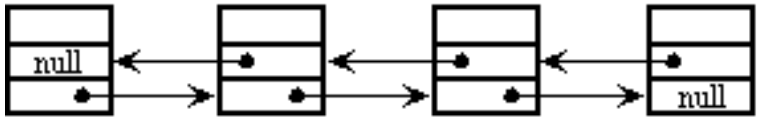
```
if ( emp.supervisor == null ) {
   System.out.println( emp.name + " is the boss!" );
} else {
   Employee runner;  // For "running" up the chain of command.
   runner = emp.supervisor;
   if ( runner.supervisor == null) {
        System.out.println( emp.name
                + " reports directly to the boss." );
   } else {
        int count = 0;
        while ( runner.supervisor != null ) {
           count++;  // Count
           runner = runner.supervisor;
        }
        System.out.println( "There are " + count
   + " supervisors between " + emp.name  + " and the boss." );
    }
}
```

Linked lists

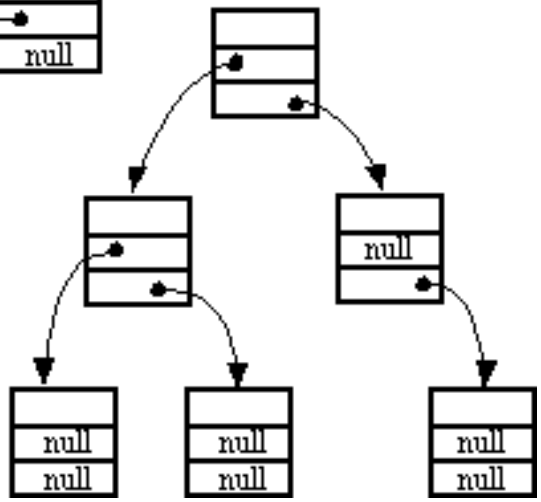
```
class Node {
  String item;
  Node next;
}
```

When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object in the list refers to the next.

```
class Node {
  String item;
  Node previous;
  Node next;
}
```



Things get even more interesting when an object contains two references to objects of the same type. In that case, more compliated data structures can be constructed.



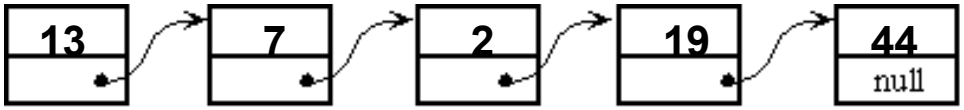
Why are linked lists so important? Arrays vs lists.

- Could we in principle always use nested arrays (one array inside another?) instead of linked lists?

Array representation

a[0]=13; a[1]=7; a[2]=2; a[3]=19; a[4]=44;

Linked list representation



When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object in the list refers to the next.

Above: two different representations of a list of elements

(13,7,2,19,44)

Code example 1: convert array to list

- Let us write this now

Code example 2: convert list to array

- Let us write this now

List vs array: unlimited / limited size (1)

- It is always possible to replace an array with a list.
- However, you cannot always replace a list by an array!

- Why: arrays have a fixed length, while lists can have arbitrary, unbounded length.
- Example from homework 2:

  - Reading in a file, consisting of rows
  - Since file length initially not known, cannot allocate a right size array
  - However, one could first build **a list of rows**, and only then convert to an array!

List vs array: access speed (2)

- Arrays have exactly one advantage, on the other hand:
  - Accessing N-th element is very fast, even in N is big.
  - Not so for lists!

List vs array: adding/deleting an element (3)

- Lists have yet another advantage:
  - Adding a new element is fast.
  - Deleting an existing element is fast.
  - Not so for arrays!

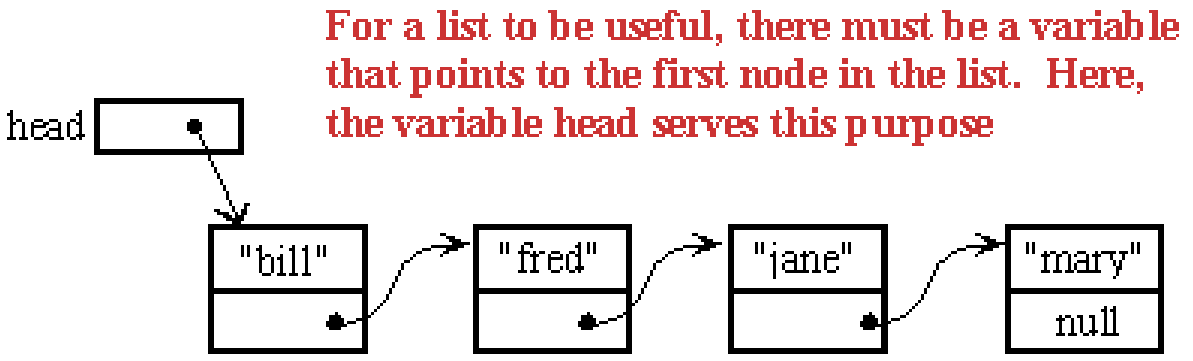
List vs array: a summary

- Advantages/disadvantages

Array		List	
Limited size	-	Unlimited size	+
Fast access	+	Slow access	-
Slow add/delete	-	Fast acces/delete	+

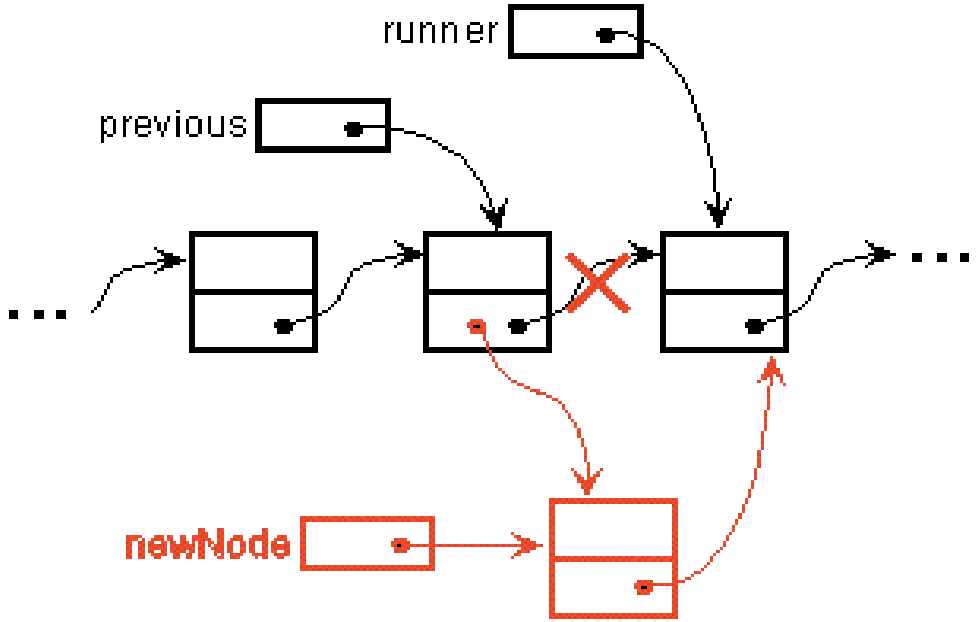
Linked lists: iterative find

```
public boolean find(String searchItem, Node head) {
   Node runner;    // A pointer for traversing the list.
   runner = head;  // Start by looking at the head of the list.
   while ( runner != null ) {
      if ( runner.item.equals(searchItem) )  return true;
      runner = runner.next;  // Move on to the next node.
   }
   return false;
} // end find()
```



Linked lists: insertion

- To insert a new item into the list:
  - Find the place in list, keep previous node in memory
  - Construct a new node
  - Let the old previous node point to new node
  - Let new node point to next node



Inserting a new node
into the middle of a list.

Linked lists: insertion code

```
public void insert(String insertItem) {
  Node newNode;           // A Node to contain the new item.
  newNode = new Node();
  newNode.item = insertItem;  // (N.B.  newNode.next is null.)
  if ( head == null ) {
     head = newNode;
  } else if ( head.item.compareTo(insertItem) >= 0 ) {
     newNode.next = head;
     head = newNode;
  } else {
     Node runner;     // A node for traversing the list.
     Node previous;   // points to the node preceding runner.
     runner = head.next;   // look at the SECOND position.
     previous = head;
     while (runner != null && runner.item.compareTo(insertItem) < 0) {
        previous = runner;
        runner = runner.next;
     }
     newNode.next = runner;     // Insert newNode after previous.
     previous.next = newNode;
   }
}  // end insert()
```

Abstract data types (ADT-s)

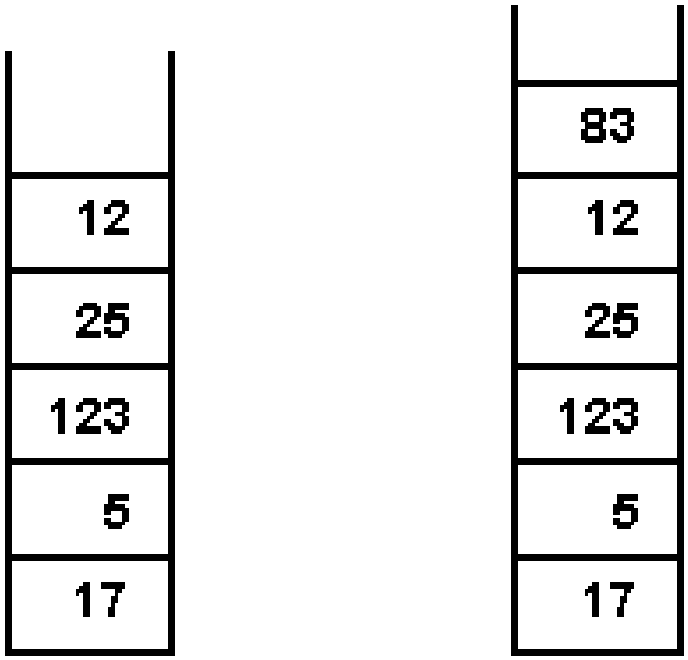
- **Java and other programming languages give us:**
  - Primitive data types like integers, floats, etc
  - Arrays
  - Structures or classes which can be used for lists etc

- **But what about other possible data types: stacks, queues, etc?**

- **We can build them ourselves**. We need to give:
  - Class definitions for keeping data
  - Methods of a class for manipulating that data (example: add a new element, delete a new element, etc)
  - If we define such a class, it is called an **abstract data type**.

Stack

In a stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the stack. The "pop" operation removes the item on the top of the stack and returns it.



Before push(83) After push(83)

Stack

- A stack consists of a sequence of items, which should be thought of piled one on top of the other like a physical stack of boxes or cafeteria trays. Only the top item on the stack is accessible at any given time.

- Three operations necessary:

  - Top element can be removed from the stack with an operation called **pop**. An item lower down on the stack can only be removed after all the items on top of it have been popped off the stack.
  - A new item can be added to the top of the stack with an operation called **push**. We can make a stack of any type of items. If, for example, the items are values of type int, then the push and pop operations can be implemented as instance methods
  - We can check if a stack is empty: implemented as a method **isempty**.

Implementation 1: StackOfInts class for integers

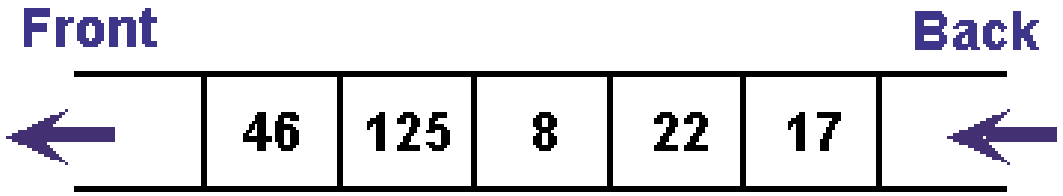
```
public class StackOfInts {
      private static class Node {
          int item;
          Node next;
       }
       private Node top; // global var holding a pointer to top
       public void push( int N ) {
          Node newTop;           // A Node to hold the new item.
          newTop = new Node();
          newTop.item = N;       // Store N in the new Node.
          newTop.next = top;     // The new Node points to the old top.
          top = newTop;          // The new item is now on top.
       }
       public int pop() {
          int topItem = top.item;  // The item that is being popped.
          top = top.next;  // The previous second item is now on top.
          return topItem;
       }
       public boolean isEmpty() { // check if stack is empty
          return (top == null);
       }
    } // end class StackOfInts
```

Another implementation: using arrays for stack

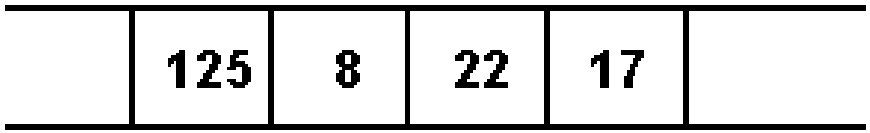
```
public class StackOfInts {
  private int[] items = new int[10];
  private int top = 0;  // The number of items currently on the stack.

  public void push( int N ) {
          if (top == items.length) {
              int[] newArray = new int[ 2*items.length ];
              System.arraycopy(items, 0, newArray, 0, items.length);
              items = newArray;
          }
          items[top] = N;  // Put N in next available spot.
          top++;           // Number of items goes up by one.
    }
    public int pop() {
          int topItem = items[top - 1]  // Top item in the stack.
          top--;    // Number of items on the stack goes down by one.
          return topItem;
    }
    public boolean isEmpty() {
          return (top == 0);
    }
} // end class
```

Queue

- A In a queue, all operations take place at one end of the queue or the other. The "enqueue" operation adds an item to the "back" of the queue. The "dequeue" operation removes the item at the "front" of the queue and returns it.



Items enter queue at back and leave from front.



After dequeue()

After enqueue(83)

Queue implementation using lists

```
public class QueueOfInts {
    private static class Node {
       int item;
       Node next;
    }
    private Node head = null;
    private Node tail = null;
    void enqueue( int N ) {
        Node newTail = new Node();
       newTail.item = N;
       if (head == null) {
          head = newTail;
          tail = newTail;
       }
       else {
          tail.next = newTail;
          tail = newTail;
       }
    }
    int dequeue() {
       int firstItem = head.item;
       head = head.next;  // The previous second item is now first.
       if (head == null) tail = null;
       return firstItem;
    }
    boolean isEmpty() {
       return (head == null);
    }
} // end class QueueOfInts
```

A few words about data types and generics

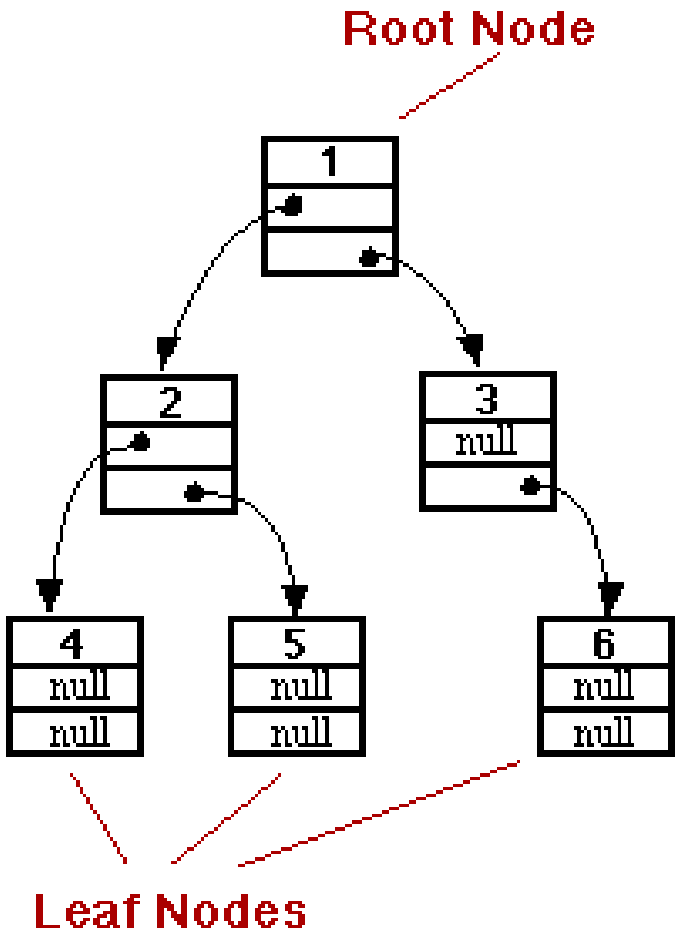
- The stack and queue examples were stacks and queues of integers.
- We could implement stack and queue also for:
  - Floats
  - Strings
  - Arrays of ints
  - Etc etc ...
- And all of these implementations would be very similar.
- Typically, a better idea is to implement one general stack (or queue) for all data types.
  - In Java, every class has class Object as a top level parent.
  - Let us make a stack and queue of elements of type Object!
  - However, primitive data types (ints, floats, etc) are not objects.
  - But, we have wrapper classes for that: Integer x = new Integer(10);
- Java 1.5 contains a special "generics" mechanism

Trees

- On the blackboard

Binary trees: recursive node counting example

```
class TreeNode {
      int item;
      TreeNode left;
      TreeNode right;
}


static int countNodes( TreeNode root ) {
  if ( root == null )
    return 0;
  else {
    int count = 1;
    count += countNodes(root.left);
    count += countNodes(root.right);
    return count;
  }
} // end countNodes
```



Binary trees as a universal datatype

- Binary trees can be used to construct any other datatype, including lists, arrays etc.
- However, often it is faster and more efficient to use other datatypes directly, not through simulating them via binary trees.
- Lisp and scheme: two datatypes used: binary trees (with list syntax) and arrays.
- On blackboard: construct the following list as a binary tree:

(a (1 2) () ((e) f))

Efficiency of standard operations on data structures

- **Unsorted array:**
  - Search is slow (every element is looked through)
  - Insertion is slow (all elements to the right must be moved)
- **Sorted array:**
  - Search is **fast** (use binary search!)
  - Insertion is slow (all elements to the right must be moved)

- **List:**
  - Search is slow (every element is looked through)
  - Insertion is **fast** (elements to the right are not moved)
- **Binary (balanced) sort tree:**
  - Search is **fast** (binary search)
  - Insertion is **fast** (elemetns to the right are not moved)